# **Vue属性**

首先用js的new关键字实例化一个vue。

* el:vue组件或对象装载在页面的位置，可通过id或class或标签名。
* template:装载的内容。HTML代码/包含指令或者其他组件的HTML片段，template将是我们使用的模板。
* data:数据通过data引入到组件中，在组件中的data要以函数的形式返回数据，当不同的界面用了同一个组件时，才不会因为一个组件的值发生改变而改变其他页面的内容。

data(){

return{

......

}

}

* computed计算属性，用于简单的运算，会保留缓存，当相关依赖没有发生改变时多次访问computed中的函数时，立即返回之前的计算结果，不会再次调用函数获得数据。
* methods属性，方法/函数没有缓存，每访问一次调用一次函数。
* watch监听vue的变量，每当变量被改动的时候都去执行特定的操作。  
  <inputtype="text"v-model="myVal">

exportdefault{

components:{

componentA:componentA

},

data(){

return{

myVal:''

}

}

watch:{

myVal:function(val,oldVal){

console.log(val,oldVal)

}

}

}

在文本框中随意输入123abd的过程中，控制台输出了文本框中的当前值和修改前的值。



* props传递数据，让子组件使用父组件的数据。  
  接受两种类型的值。  
  ①数组props:['first','sceond',...]  
  ②对象

props:{

'number-to-do':Number,

'second':[Number,String,Object]

}

# Vue-methods方法

### 一、methods中参数的传递

使用方法和正常的javascript传递参数的方法一样，分为两部：

1、在methods的方法中进行声明，比如我们给add方法加上一个num参数，就要写出add:function(num){}.

2、调用方法时直接传递，比如我们要传递2这个参数，我们在button上就直接可以写。<button@click=”add(2)”></button>.

|  |
| --- |
| <!DOCTYPE html>  <html lang="en">  <head>  <meta charset="UTF-8">  <script type="text/javascript" src="../assets/js/vue.js"></script>  <title>methods Option</title>  </head>  <body>  <h1>methods Option</h1>  <hr>  <div id="app">  {{a}}  <p><button @click="add(2)">add</button></p>  </div>    <script type="text/javascript">  var app=new Vue({  el:'#app',  data:{  a:1  },  methods:{  add:function(num){  if(num!=''){this.a+=num}  else{this.a++}  }  }  })  </script>  </body>  </html> |

### 二、methods中的$event参数

传递的$event参数都是关于你点击鼠标的一些事件和属性。下面是传递的方法。

传递：<button@click=”add(2,$event)”>add</button>。

### 三、native给组件绑定构造器里的原生事件

在实际开发中经常需要把某个按钮封装成组件，然后反复使用，如何让组件调用构造器里的方法，而不是组件里的方法。就需要用到.native修饰器。

现在把add按钮封装成组件：

声明btn对象：

|  |
| --- |
| varbtn={  template:`<button>组件Add</button>`  } |

在构造器里声明：

|  |
| --- |
| components:{  "btn":btn  } |

用.native修饰器来调用构造器里的add方法

|  |
| --- |
| <p><btn@click.native="add(3)"></btn></p> |

### 四、作用域外部调用构造器里的方法

这种不经常使用，如果出现了这种情况，说明代码组织不够好。

|  |
| --- |
| <buttononclick="app.add(4)">外部调用构造器里的方法</button> |

# **Vue 双向绑定原理**

数据绑定是让数据 M（model）展示到视图 V（view）上。目前前端框架基本上都是采用 MVVM 模式实现双向绑定，Vue不例外。但是各个框架实现双向绑定的方法略有所不同，目前大概有三种实现方式。

* 发布订阅模式
* Angular 的脏查机制
* 数据劫持

而 Vue 则采用的是数据劫持与发布订阅相结合的方式实现双向绑定，数据劫持主要通过Object.defineProperty来实现。

#### Object.defineProperty

Object.defineProperty的用法，主要看看它的存储属性 get 与 set。

var people = {

name: "Modeng",

age: 18

}

people.age; //18

people.age = 20;

上述代码是普通的获取/设置对象的属性。

var modeng = {}

var age;

Object.defineProperty(modeng, 'age', {

get: function () {

console.log("获取年龄");

return age;

},

set: function (newVal) {

console.log("设置年龄");

age = newVal;

}

});

modeng.age = 18;

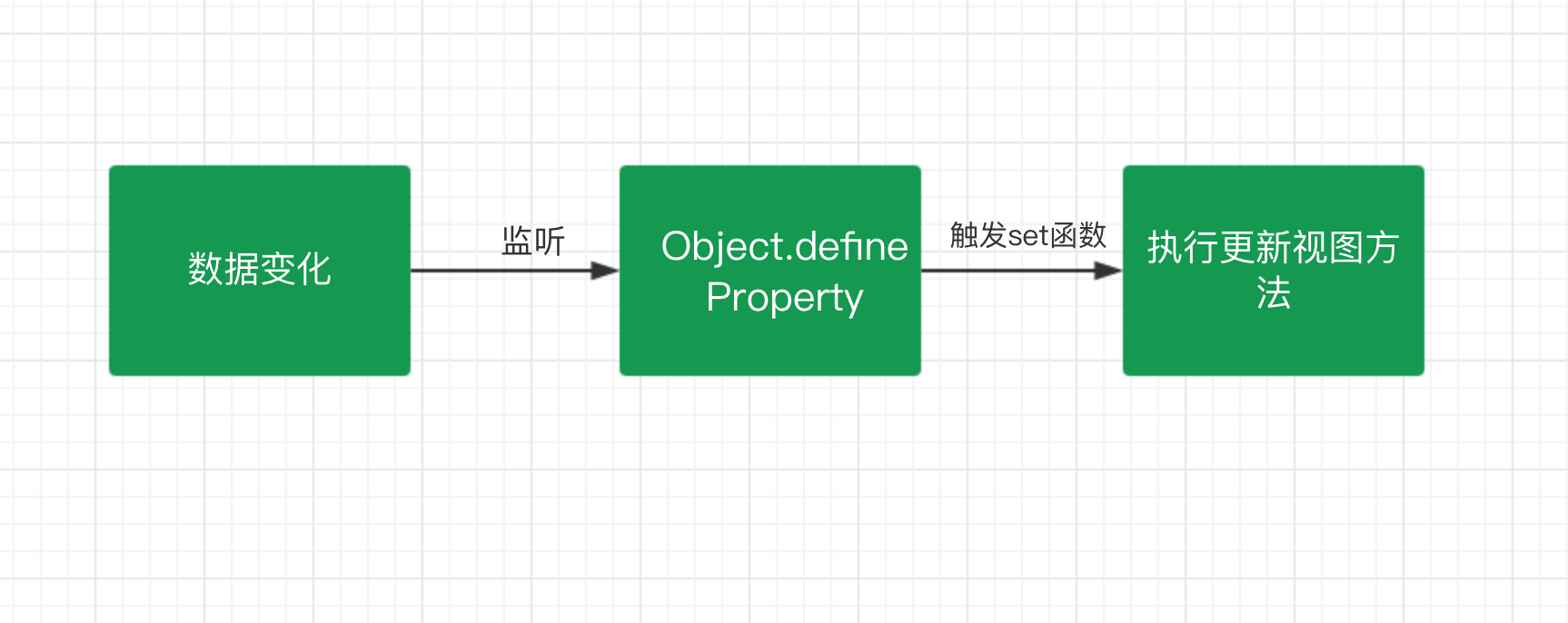
console.log(modeng.age);

通过上述操作之后，我们访问 age 属性时会自动执行 get 函数，设置 age 属性时，会自动执行 set 函数，这就给我们的双向绑定提供了非常大的方便。

#### 分析

MVVM 模式在于数据与视图的保持同步，意思是说数据改变时会自动更新视图，视图发生变化时会更新数据。

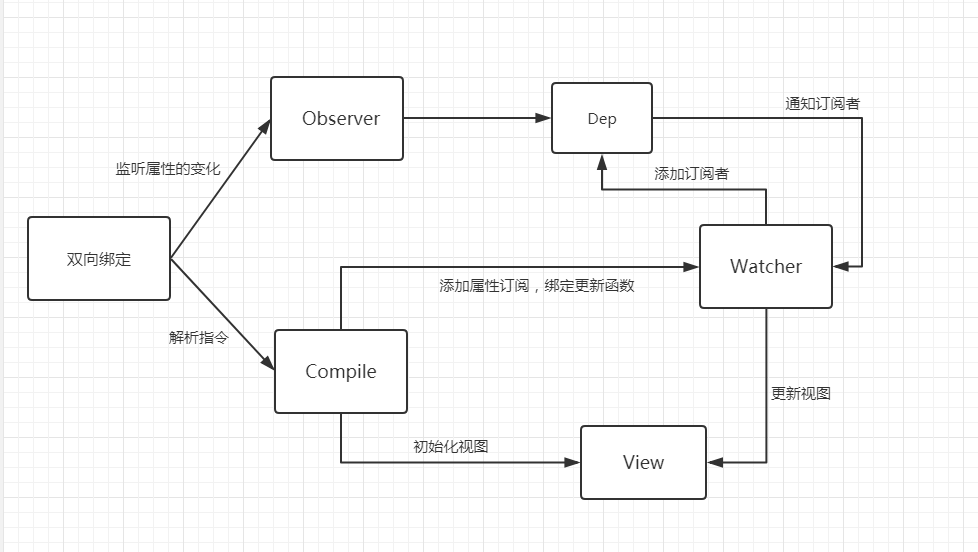
所以我们需要做的就是如何检测到数据的变化然后通知我们去更新视图，如何检测到视图的变化然后去更新数据。检测视图这个比较简单，无非就是我们利用事件的监听即可。利用Object.defineProperty可以做到当我们的属性发生变化时，它会自动触发 set 函数从而能够通知我们去更新视图。



#### 实现

通过上面的描述与分析我们知道 Vue 是通过数据劫持结合发布订阅模式来实现双向绑定的。我们也知道数据劫持是通过Object.defineProperty方法。当我们知道这些之后，我们就需要一个监听器 Observer 来监听属性的变化。得知属性发生变化之后我们需要一个 Watcher 订阅者来更新视图，我们还需要一个 compile 指令解析器，用于解析我们的节点元素的指令与初始化视图。所以我们需要如下：

* Observer 监听器：用来监听属性的变化通知订阅者
* Watcher 订阅者：收到属性的变化，然后更新视图
* Compile 解析器：解析指令，初始化模版，绑定订阅者



顺着这条思路我们一步一步去实现。

#### 监听器 Observer

监听器的作用就是去监听数据的每一个属性，我们上面也说了使用Object.defineProperty方法，当我们监听到属性发生变化之后我们需要通知 Watcher 订阅者执行更新函数去更新视图，在这个过程中我们可能会有很多个订阅者 Watcher 所以我们要创建一个容器 Dep 去做一个统一的管理。

function defineReactive(data, key, value) {

//递归调用，监听所有属性

observer(value);

var dep = new Dep();

Object.defineProperty(data, key, {

get: function () {

if (Dep.target) {

dep.addSub(Dep.target);

}

return value;

},

set: function (newVal) {

if (value !== newVal) {

value = newVal;

dep.notify(); //通知订阅器

}

}

});

}

function observer(data) {

if (!data || typeof data !== "object") {

return;

}

Object.keys(data).forEach(key => {

defineReactive(data, key, data[key]);

});

}

function Dep() {

this.subs = [];

}

Dep.prototype.addSub = function (sub) {

this.subs.push(sub);

}

Dep.prototype.notify = function () {

console.log('属性变化通知 Watcher 执行更新视图函数');

this.subs.forEach(sub => {

sub.update();

})

}

Dep.target = null;

以上我们就创建了一个监听器 Observer，我们现在可以尝试一下给一个对象添加监听然后改变属性会有何变化。

var modeng = {

age: 18

}

observer(modeng);

modeng.age = 20;

我们可以看到浏览器控制台打印出 “属性变化通知 Watcher 执行更新视图函数” 说明我们实现的监听器没毛病，既然监听器有了，我们就可以通知属性变化了，那肯定是需要 Watcher 的时候了。

#### 订阅者 Watcher

Watcher 主要是接受属性变化的通知，然后去执行更新函数去更新视图，所以我们做的主要是有两步：

* 把 Watcher 添加到 Dep 容器中，这里我们用到了 监听器的 get 函数
* 接收到通知，执行更新函数。

function Watcher(vm, prop, callback) {

this.vm = vm;

this.prop = prop;

this.callback = callback;

this.value = this.get();

}

Watcher.prototype = {

update: function () {

const value = this.vm.$data[this.prop];

const oldVal = this.value;

if (value !== oldVal) {

this.value = value;

this.callback(value);

}

},

get: function () {

Dep.target = this; //储存订阅器

const value = this.vm.$data[this.prop]; //因为属性被监听，这一步会执行监听器里的 get方法

Dep.target = null;

return value;

}

}

这一步我们把 Watcher 也给弄了出来，到这一步我们已经实现了一个简单的双向绑定了，我们可以尝试把两者结合起来看下效果。

function Mvue(options, prop) {

this.$options = options;

this.$data = options.data;

this.$prop = prop;

this.$el = document.querySelector(options.el);

this.init();

}

Mvue.prototype.init = function () {

observer(this.$data);

this.$el.textContent = this.$data[this.$prop];

new Watcher(this, this.$prop, value => {

this.$el.textContent = value;

});

}

这里我们尝试利用一个实例来把数据与需要监听的属性传递进来，通过监听器监听数据，然后添加属性订阅，绑定更新函数。

<div>{{name}}</div>

const vm = new Mvue({

el: "#app",

data: {

name: "我是摩登"

}

}, "name");

我们可以看到数据已经正常的显示在页面上，那么我们在通过控制台去修改数据，发生变化后视图也会跟着修改。

到这一步我们我们基本上已经实现了一个简单的双向绑定，但是不难发现我们这里的属性都是写死的，也没有指令模板的解析，所以下一步我们来实现一个模板解析器。

#### Compile 解析器

Compile 的主要作用一个是用来解析指令初始化模板，一个是用来添加添加订阅者，绑定更新函数。

因为在解析 DOM 节点的过程中我们会频繁的操作 DOM， 所以我们利用文档片段（DocumentFragment）来帮助我们去解析 DOM 优化性能。

function Compile(vm) {

this.vm = vm;

this.el = vm.$el;

this.fragment = null;

this.init();

}

Compile.prototype = {

init: function () {

this.fragment = this.nodeFragment(this.el);

},

nodeFragment: function (el) {

const fragment = document.createDocumentFragment();

let child = el.firstChild;

//将子节点，全部移动文档片段里

while (child) {

fragment.appendChild(child);

child = el.firstChild;

}

return fragment;

}

}

然后我们就需要对整个节点和指令进行处理编译，根据不同的节点去调用不同的渲染函数，绑定更新函数，编译完成之后，再把 DOM 片段添加到页面中。

Compile.prototype = {

compileNode: function (fragment) {

let childNodes = fragment.childNodes;

[...childNodes].forEach(node => {

let reg = /\{\{(.\*)\}\}/;

let text = node.textContent;

if (this.isElementNode(node)) {

this.compile(node); //渲染指令模板

} else if (this.isTextNode(node) && reg.test(text)) {

let prop = RegExp.$1;

this.compileText(node, prop); //渲染{{}} 模板

}

//递归编译子节点

if (node.childNodes && node.childNodes.length) {

this.compileNode(node);

}

});

},

compile: function (node) {

let nodeAttrs = node.attributes;

[...nodeAttrs].forEach(attr => {

let name = attr.name;

if (this.isDirective(name)) {

let value = attr.value;

if (name === "v-model") {

this.compileModel(node, value);

}

node.removeAttribute(name);

}

});

},

//省略。。。

}

现在我们实现了 Observer、Watcher、Compile，接下来就是把三者给组织起来，成为一个完整的 MVVM。

#### 创建 Mvue

这里我们创建一个 Mvue 的类（构造函数）用来承载 Observer、Watcher、Compile 三者。

function Mvue(options) {

this.$options = options;

this.$data = options.data;

this.$el = document.querySelector(options.el);

this.init();

}

Mvue.prototype.init = function () {

observer(this.$data);

new Compile(this);

}

然后我们就去测试一下结果，看看我们实现的 Mvue 是不是真的可以运行。

<div>

<h1>{{name}}</h1>

</div>

<script src="./js/observer.js"></script>

<script src="./js/watcher.js"></script>

<script src="./js/compile.js"></script>

<script src="./js/index.js"></script>

<script>

const vm = new Mvue({

el: "#app",

data: {

name: "完全没问题，看起来是不是很酷！"

}

});

</script>

#### 数据代理

我们来改造下 Mvue 添加数据代理功能，我们也是利用Object.defineProperty方法进行一步中间的转换操作，间接的去访问。

function Mvue(options) {

this.$options = options;

this.$data = options.data;

this.$el = document.querySelector(options.el);

//数据代理

Object.keys(this.$data).forEach(key => {

this.proxyData(key);

});

this.init();

}

Mvue.prototype.init = function () {

observer(this.$data);

new Compile(this);

}

Mvue.prototype.proxyData = function (key) {

Object.defineProperty(this, key, {

get: function () {

return this.$data[key]

},

set: function (value) {

this.$data[key] = value;

}

});

}

到这里我们就可以像 Vue 一样去修改我们的属性了。

# vue中关于 $emit 的用法

1、父组件可以使用 props 把数据传给子组件。  
2、子组件可以使用 $emit 触发父组件的自定义事件。

vm.$emit(event, arg) // 触发当前实例上的事件

vm.$on(event, fn);// 监听 event 事件后运行 fn；

例如：子组件：

<template>

<div>

<h3>父组件传给子组件的toCity:{{sendData}}</h3>

<br/><button @click='select(`大连`)'>点击此处将‘大连’发射给父组件</button>

</div>

</template>

<script>

export default {

name:'trainCity',

props:['sendData'], // 用来接收父组件传给子组件的数据

methods:{

select(val) {

let data = {

cityname: val

};

this.$emit('showCityName',data);//select事件触发后，自动触发showCityName事件

}

}

}

</script>

<template>

<div>

<h3>父组件传给子组件的toCity:{{sendData}}</h3>

<br/><button @click='select(`大连`)'>点击此处将‘大连’发射给父组件</button>

</div>

</template>

<script>

export default {

name:'trainCity',

props:['sendData'], // 用来接收父组件传给子组件的数据

methods:{

select(val) {

let data = {

cityname: val

};

this.$emit('showCityName',data);//select事件触发后，自动触发showCityName事件

}

}

}

</script>

父组件：

<template>

<div>

<div>父组件的toCity{{toCity}}</div>

<train-city @showCityName="updateCity" :sendData="toCity"></train-city>

</div>

<template>

<script>

import TrainCity from "./train-city";

export default {

name:'index',

components: {TrainCity},

data () {

return {

toCity:"北京"

}

},

methods:{

updateCity(data){//触发子组件城市选择-选择城市的事件

this.toCity = data.cityname;//改变了父组件的值

console.log('toCity:'+this.toCity)

}

}

}

</script>

<template>

<div>

<div>父组件的toCity{{toCity}}</div>

<train-city @showCityName="updateCity" :sendData="toCity"></train-city>

</div>

<template>

<script>

import TrainCity from "./train-city";

export default {

name:'index',

components: {TrainCity},

data () {

return {

toCity:"北京"

}

},

methods:{

updateCity(data){//触发子组件城市选择-选择城市的事件

this.toCity = data.cityname;//改变了父组件的值

console.log('toCity:'+this.toCity)

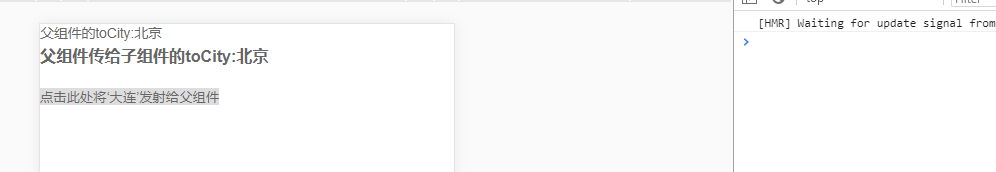
}

}

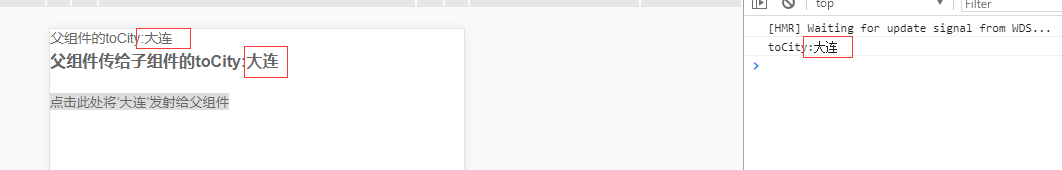
}

</script>

图一：点击之前的数据



图二：点击之后的数据



# Vue 中的 computed 属性

* computed 用来监控自己定义的变量，该变量不在 data 里面声明，直接在 computed 里面定义，然后就可以在页面上进行双向数据绑定展示出结果或者用作其他处理；
* computed 比较适合对多个变量或者对象进行处理后返回一个结果值，也就是数多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化，举例：购物车里面的商品列表和总金额之间的关系，只要商品列表里面的商品数量发生变化，或减少或增多或删除商品，总金额都应该发生变化。这里的这个总金额使用 computed 属性来进行计算是最好的选择

与 watch 之间的区别：

* watch 主要用于监控 vue 实例的变化，它监控的变量当然必须在 data 里面声明才可以，它可以监控一个变量，也可以是一个对象，但是我们不能类似这样监控，比如：

watch:{

goodsList.price(newVal,oldVal){

//监控商品列表中是商品价格

}

}

这样会报错。只能监控整个对象或单个变量，如下所示：

　data(){

　　　　　　　　return {

　　　　　　　　　　example0:"",

　　　　　　　　　　example1:"",

　　　　　　　　　　example2:{

　　　　　　　　　　　　inner0:1,

　　　innner1:2

　}

　　　　　　}

　　　　},

watch:{

　example0(newVal,oldVal){//监控单个变量

……

}，

example2(newVal,oldVal){//监控对象

……

}，

}

* watch 一般用于监控路由、input 输入框的值特殊处理等等，它比较适合的场景是一个数据影响多个数据

# 生命周期函数

实例生命周期钩子

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做生命周期钩子的函数，这给了用户在不同阶段添加自己的代码的机会。

比如 created 钩子可以用来在一个实例被创建之后执行代码：

new Vue({

data: {

a: 1

},

created: function () {

// `this` 指向 vm 实例

console.log('a is: ' + this.a)

}

})

// => "a is: 1"

也有一些其它的钩子，在实例生命周期的不同阶段被调用，如 mounted、updated 和 destroyed。生命周期钩子的 this 上下文指向调用它的 Vue 实例。

不要在选项属性或回调上使用箭头函数，比如 created: () => console.log(this.a) 或 vm.$watch('a', newValue => this.myMethod())。因为箭头函数并没有 this，this 会作为变量一直向上级词法作用域查找，直至找到为止，经常导致 Uncaught TypeError: Cannot read property of undefined 或 Uncaught TypeError: this.myMethod is not a function 之类的错误。

生命周期图示如下图：

